Why Functional Programming Matters

为什么函数式编程至关重要*

作者：John Hughes

翻译：赵常坤

摘要

随着软件越来越复杂，良好的组织结构就变得越来越重要。“结构良好”的软件容易编写，容易调试并且提供一系列可以重用的模块来降低未来编程的成本。传统的编程语言在将问题模块化时有很多概念上的限制，函数式语言清除了这些限制。本文特别展示了函数式编程语言的两个特点，高阶函数和惰性求值，它们在模块化方面有重要作用。作为例子，我们操作列表和树，编写几个数值算法，并且实现了alpha-beta启发式算法（一种被应用于游戏编程的人工智能算法）。由于模块化是编程成功的金钥匙，所以函数式编程在现实中相当重要。

1 简介

本文试图证明在现实中函数式编程是及其重要的，同时帮助函数程序员弄清楚并且最大限度地开发利用函数式编程的优点。

之所以称之为函数式编程是因为一个程序包含的全是函数。主函数本身被写成一个函数，它接受程序输入作为参数并且提交程序输出作为其结果。通常主

*本文作于1984年，作为Chalmers大学的备忘录流传了几年。轻微修订的版本出现于1989年[Hug89]和1990年[Hug90]。这个版本基于原始的Chalmers大学备忘录nroff格式的源代码，针对LaTeX做了一点修改以使其更接近发行版，并且修正了一两处错误。请原谅这种过时的排版吧，另外文中的例子不是用Haskell写的

函数是以其他函数定义的，这些函数也是以其他函数定义的……直到最底层：函数都是语言的原语，这些函数和普通的数学函数非常像，在本文中将使用普通等式定义。本文的标记法遵循Turner的编程语言Miranda(TM),但是之前没有函数式编程语言基础也可以看懂。（Miranda是Research Software Ltd.的注册商标）

函数式编程的特点和优势通常被概括如下。函数式编程没有赋值语句，因此变量给定一个值后就不允许再改变。更一般的，函数式编程完全没有副作用。函数除了计算自己的结果没有任何其他作用。这消除了一个主要Bug滋生源，同时使计算顺序变得无关-因为没有副作用可以改变一个表达式的值，因此可以在任何时候对其求值。这就把程序员从指定程序流程的担子里解脱出来。因为表达式可以在任何时间求值，所以可以自由地使用值代替变量，反之亦然—就是说，程序是“引用透明”的。这就使函数式编程在数学上更容易处理。

这一堆的优点都不错，但是如果有外行对其不以为然也不足为奇。它说了太多的函数式编程不是什么（没有赋值，没有副作用，没有控制流）但是却没怎么涉及它是什么。函数式编程的程序员听起来像古代的和尚，拒绝安逸的生活并希望以此使自己变得高尚。对物质利益越感兴趣的人，越不把这些优点当回事。

函数式程序员争论说其实函数式程序设计存在着巨大的物质利益—函数式程序员拥有比同行高的生产力，因为函数式的程序更加短小精悍。为什么会这样？基于上述的优点，唯一靠不住的说法就是传统的程序中包含了90% 的赋值语句，而在函数式程序中它们是可以避免的！这简直是扯淡。如果省略赋值语句会带来这么大的好处，那么FORTRAN程序员早在20年前就这么做了。省略一些特性会使语言变得更强大，即使这些特性再烂，这在逻辑上也是不通的。

甚至是函数式程序员都会对这些所谓的优点感到不满，因为它们对于程序员们发掘函数式编程的威力毫无帮助。不能写刻意不使用赋值或者刻意引用透明的程序。这里没有程序质量的标准，因此也不存在完美的目标。

显然这样描述函数式编程是不够充分的。我们必须找到可以恰如其分地解释函数式编程强大威力的东西，同时给函数式程序员一个清晰的奋斗目标。

2 与结构化程序设计的类比

将函数式编程和结构化编程做一个类比是很有好处的。过去，结构化编程的特点和优点大致概括如下：结构化程序不包含goto语句；结构化程序中的块没有多个入口和出口；结构化的程序比非结构化的程序在数学上更可控。这些优点和我们前面讨论的函数式编程的优点在本质上很相似。他们用的本质上都是否定语气，并且引发了关于“goto语句是否必要”的徒劳争论。

从事后看，结构化编程的这些属性，尽管很有用，但是也还没有触及问题的核心。结构化和非结构化程序的最重要的不同是前者是按照模块化的方式设计的。模块化设计带来了生产力的巨大提高。首先，小模块可以快速并提早进行编码；其次，通用目的的模块可以重用，带来了后续程序的快速开发；再次，程序的模块可以单独测试，减少了程序调试时间。

没有goto等等影响不大，它有助于“小规模编程”，而模块化设计有助于“大规模编程”，因此程序员在FORTRAN或者汇编语言中都可出享受结构化编程的好处，即使需要多做一点工作。

现在，模块化是成功编程的关键的观点已经被普遍接受，一些语言，像Modula-II[Wir82]，Ada[oD80]和Standard ML[MTH90]都包含了特别为改进模块化而设计的特性。但是有一个重点经常被忽略。写模块化程序的时候，首先将问题分解为子问题，然后解决这些子问题并将他们组合起来。程序员能以什么方式分解原始问题直接取决于他们如何能够把解决方案粘起来。所以，为了在概念上增强程序员模块化编程的能力，必须在编程语言中提供新的黏合剂。复杂的规则和对分块编译的支持仅对文本层面的细节有帮助，它们没有曾概念上提供分解问题的新工具。

通过和木工对比可以充分意识到黏合剂的重要性：通过早好坐垫、腿和靠背等等然后把他们按照正确的方式钉起来可以很容易的造出椅子来。但是这取决于接合木板的能力，没有这个，要造椅子只能用一整块木头雕刻出来，这要困难得多。这个例子说明了模块化编程的巨大威力以及正确的黏合剂的重要性。

现在让我们转向函数式编程。在本文剩余的部分，我们会证明函数式编程语言提供了两种新的、非常重要的黏合剂。我们会给出很多的例子程序，它们可以按照新的方法进行模块化设计，藉此变得更加简介。这是函数式编程的威力所在—它允许极大的改进模块化设计。这也是函数式程序员必须努力追求的

目标—更小、更简捷、更通用的模块，使用我们将要描述的新的黏合剂粘在一起。

3 把函数粘到一起

两种粘合剂中的第一种能够把简单函数黏合成一个更加复杂的函数。可以以一个简单的列表处理过程展示— 累加列表中的元素，我们这样定义列表:

$$\text{listof X} ::= \text{nil} \mid \text{cons X (listof X)}$$

表示一个列表Xs[1] (不管X是什么东东) 或者是nil，代表空表，或者是一个X和另外一个列表Xs的一个cons。cons代表一个列表，它的第一个元素是X，第二个及后续元素是另外一个列表Xs的元素。这里X可以代表任何类型—举个例子，如果X是“integer”，那么这个定义就是：一个由整数组成的列表或者是空表或者是一个整数和另外一个整数列表的cons。按照时间中通常的方法，我们写列表时只是简单的由方括号包围列表元素，而不是显式地写出cons和nil。这简化了符号的书写，例如：

[ ]	means	nil
[1]	means	cons 1 nil
[1, 2, 3]	means	cons 1 (cons 2 (cons 3 nil))

列表元素可以通过递归函数*sum*累加起来。sum必须被定义成能接受两种参数：一个空表（nil），一个cons。因为没有数字的和为零，所以我们定义

$$\text{sum nil} = 0$$

累加一个cons可以看作是列表的一个元素加上列表其他元素的和，因此我们可以定义：

$$\text{sum (cons num list)} = \text{num} + \text{sum list}$$

仔细检查这个定义我们就可以发现只有下面盒子里的部分是特定于求和运算的：

[1]用Xs来表示由X组成的列表—译者

$$\text{sum nil} = \boxed{\quad 0 \quad}$$

$$\text{sum (cons num list)} = \text{num} \ \boxed{\quad + \quad} \ \text{sum list}$$

这意味着sum的计算可以通过黏合一个递归模式和上面盒子里的部分来实现模块化。这个递归模式通常被称作reduce，sum可以这样表示：

sum = reduce add 0

方便起见，reduce 传入一个函数add而不是运算符，add定义为：

add x y = x + y

reduce的定义仅从参数化sum定义就可以得到，我们给出：

(reduce f x) nil = x
(reduce f x) (cons a l) = f a ((reduce f x) l)

这里我们用小括号把(reduce f x) 括起来来更清晰地表示它代替sum。通常小括号是省略的，因而((reduce f x) l)写作(reduce f x l)。像reduce这样有3个参数的函数，只使用两个参数调用就得到一个以另外一个参数作为参数的函数，并且一般的说，一个有n个参数的函数应用于$m(< n)$ 个时便得到一个以剩余的$m - n$个参数为参数的函数。我们将来会遵守这个约定。

如此模块化sum，我们就得到了部分重用的好处。reduce是最有趣的部分，可以被用来写出一个将列表所有元素相乘的程序，而不需要额外编程：

product = reduce multiply 1

它也可以被用来检查boolean列表中是否由元素为true：

anytrue = reduce or false

或者是否全部都是true：

alltrue = reduce and true

把(reduce f a) 作为函数理解的一个方法是：将列表中出现cons的地方都用f代替，nil用a代替，以[1, 2, 3]为例，就是：

cons 1 (cons 2 (cons 3 nil))

(reduce add 0)就把它转换为：

add 1 (add 2 (add 3 0)) = 6

(reduce multiply 1)就把它转换为：

multiply 1 (multiply 2 (multiply 3 1)) = 6

现在很明显了，(reduce cons nil) 就是复制一个列表。因为一个列表可以通过将其每个元素cons到另外一个列表前面从而附加到另一个列表，所以我们有：

append a b = reduce cons b a

如：

append [1,2] [3,4]	=	reduce cons [3,4] [1,2]
	=	(reduce cons [3,4]) (cons 1 (cons 2 nil))
	=	(cons 1 (cons 2 [3,4]))
		(用cons代替cons，[3,4]代替nil)
	=	[1,2,3,4]

把列表所有元素都乘2的函数可以这样写：

doubleall = reduce doubleandcons nil

其中 doubleandcons num list = cons (2*num) list

doubleandcons可以进一步模块化，首先：

doubleandcons = fandcons double

其中　　double n = 2*n

　　fandcons f el list = cons (f el) list

进一步：

　　fandcons f = cons . f

其中“.”(函数复合，是标准操作符)定义为：

　　(f . g) h = f (g h)

把fandcons新的定义应用到一些参数上我们就可以看到他是正确的：

　　fandcons f el = (cons . f) el = cons (f el)

因此　　fandcons f el list = cons (f el) list

最后的版本就是：　　doubleall = reduce (cons . double) nil

使用更深入的模块化我们得到：

　　doubleall = map double

　　map f = reduce (cons . f) nil

这里map将函数f应用于列表的每个元素。map是另一个常用的函数。

我们甚至可以写一个把一个矩阵的所有元素相加的函数，矩阵使用列表的列表表示：

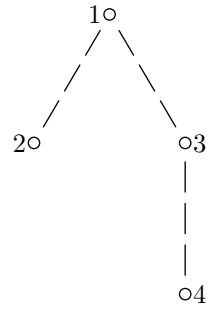
　　summatrix = sum . map sum

map sum使用sum累加每行，然后最左面的sum把各行的和累加就得到整个矩阵的和。

这些例子应该足以使读者确信，一点点模块化就可以产生巨大的效果。把一个简单函数(sum)模块化成“高阶函数”和一些简单参数的符合函数，我们就得到了一个部件(reduce)，可以用于编写很多其它列表函数而无需更多的编程努力。我们不必限于列表程序。作为另外一个例子，请考虑有序标记树，定义为：

treeof X ::= node X (listof (treeof X))

这个定义时说一颗X树是一个节点，这个节点有一个标签X和一个子树的列表，子树也是X树。比如下面这棵树



可以表示为:

```
node 1
    (cons (node 2 nil)
          (cons (node 3
                      (cons (node 4 nil) nil))
                nil))
```

我们不再举例一个函数然后从其中抽象出高阶函数，而是直接给出一个类似于reduce的函数redtree。回想一下reduce接收两个参数，一个代替cons，一个代替nil。因为tree是使用node,cons和nil定义的，所以redtree必须接收三个参数— 分别代替它们。tree和list是不同的类型，所以我们必须定义两个函数，分别作用于每个类型。所以我们定义：

```
redtree   f g a (node label subtrees) =
          f label (redtree' f g a subtrees)
redtree'  f g a (cons subtree rest) =
          g (redtree f g a subtree) (redtree' f g a rest)
redtree'  f g a nil = a
```

结合redtree和其他函数可以定义除许多有趣的程序。举个例子，把一棵树所有节点的label相加可以使用：

sumtree = redtree add add 0

以上面的树为例，sumtree就是：

```
add 1
    (add (add 2 0)
         (add (add 3
                   (add (add 4 0) 0))
              0))
 = 10
```

求一棵树所有的节点label构成的列表使用：

labels = redtree cons append nil

那个例子就得到：

```
cons 1
     (append (cons 2 nil)
             (append (cons 3
                           ((append 4 nil) nil))
                     nil))
 = [1,2,3,4]
```

最后定义一个类似map的函数用来对列表的每个label应用函数f:

maptree f = redtree (node . f) cons nil

这些之所以能够实现，因为函数式编程语言允许把在传统的语言中不可分割的函数看作是组建的组合体—一个通用的高阶函数和一些特定的函数。高阶函数一旦定义，就可以使一些操作变得非常容易编写。当定义了新的数据类型时，就要编写处理这种类型的高阶函数。这简化了对新数据类型的操作，同时把它的表示细节等相关的知识局部化了。最类似的传统语言是可扩展语言—只要有新的需求好像它就可以扩展出新的控制结构。

4 把程序粘在一起

函数式语言提供的另一种新的黏合剂可以将整个程序黏合在起来。回忆一下，一个完整的函数式程序只不过是一个从输入到输出的函数。如果f 和g是这样的两个程序，那么(g . f)就是这样一个函数：把它作用到其输入时，计算的是

g (f input)

程序f的输出被用作程序g的输入。传统上，这可能要通过把f的输出存在临时文件中来实现。问题是临时文件可能需要占用很大的内存，从而导致这样黏合程序不现实。函数式语言提供了解决之道。f和g严格同步地返回。f只有在g试图读入输入时才开始执行，并且仅仅执行到刚好输出满足了g的需要。然后f暂停g运行直到其要求另外的输入。一个额外的好处是：如果g没有读完f的所有输出就终止了，那f也会终止。f甚至可以是一个没有终止的程序，产生无限量的输出，因为它可以在f结束后被强制终止。这样就允许将终止条件和循环体分开—这是种强大的模块化。

因为这种求值方法尽可能少的运行f，所以被称为“惰性求值”。它使得把一个程序模块化成一个产生大量可能结果的生产者和一个选择合适结果的消费者便的可行。一些系统允许程序按照这种方式运行，但是只有函数式语言对每个函数调用都一律使用惰性求值，允许将程序的任何部分按照这种方式模块化。惰性求值恐怕是函数式程序员最强大的模块化工具。

4.1 Newton-Raphson平方根

我们来编写几个数值算法来展示惰性求值的威力。首先。考虑球平方根的Newton-Raphson算法。这个算法这样求N的平方根：从一个初始值a_0开始，使用下面的规则计算越来越逼近结果的值。

$$a_{n+1} = (a_n + N/a_n)/2$$

如果近似值序列收敛于极限a，那么有：

$$a = (a + N/a)/2$$

so $$2a = a + N/a$$

$$a = N/a$$

$$a_2 = N$$

$$a = squareroot(N)$$

实际上，近似值序列确实会很快收敛。平方根函数以一个误差(eps)作为参数，当两个相邻的近似值相差小于eps时停止。

这个算法通常这样编写：

```
C     N IS CALLED ZN HERE SO THAT IT HAS THE RIGHT TYPE
            X = A0
            Y = A0 + 2.*EPS
C     THE VALUE OF Y DOES NOT MATTER SO LONG AS ABS(X-Y).GT.EPS
100         IF (ABS(X-Y).LE.EPS) GOTO 200
            Y = X
            X = (X + ZN/X) / 2
200         CONTINUE
C     THE SQUARE ROOT OF ZN IS NOW IN X
```

这个程序[1] 在传统语言中是不可分的。我们将使用惰性求值将其表示为更模块化的形式。然后展示生成的部件的一些其它用途。

因为Newton-Raphson算法是计算一系列近似值，所以将程序表示为一个近似值的列表是很自然的。每个近似值都通过下面的函数由前一个近似值决定

```
next N x = (x + N/x) / 2
```

(next N)是一个将每个近似值都作用于下一个的函数。把这个函数称为f，近似值序列就是

```
[a0, f a0, f(f a0), f(f(f a0)), ...]
```

我们可以定义一个函数来计算它：

```
repeat f a = cons a (repeat f (f a))
```

于是近似值序列可以用下式计算：

[1] 这是一段FORTRAN程序，C 代表注释行（译者）

```
repeat (next N) a0
```

Repeat是一个有无限输出的函数的例子，但是这不要紧，因为不会有多于程序其它部分需要的近似值被产生出来。这里的无限仅仅是潜在意义上的：它仅仅意味着近似值如果需要就可以计算出来，重复它自身是没有限制的。

查找平方根的剩余部分是函数within，接收一个误差和近似值列表作为参数，沿着列表找到两个连续的近似值，他们的差不超过给出的误差。它可以这样定义：

```
within eps (cons a (cons b rest)) =
        = b,                          if abs(a-b) <= eps
        = within eps (cons b rest), otherwise
```

放到一起就得到：

```
sqrt a0 eps N = within eps (repeat (next N) a0)
```

现在我们有了查找平方根的部件，可以试着以不同的方式组合它们。我们要做的一个改进是用相邻两个近似值的比值与1接近程度而不是其差相对0的接近程度来判断结束。这对非常小(起始的相邻近似值的差就很小)和非常大(舍入误差都比给出的误差大很多)的数字更加合适。仅重新定义下within就可以了：

```
relative eps (cons a (cons b rest)) =
        = b,                            if abs(a-b) <= eps*abs b
        = relative eps (cons b rest), otherwise
```

现在新版本的sprt可以这样定义：

```
relativesqrt a0 eps N = relative eps (repeat (next N) a0)
```

不需要重写产生近似序列的部分。

4.2 数值微分

我们在平方根程序中重用了近似值序列产生函数。当然我们因为可以和任何产

生一个近似值序列的数值算法一起重用within和relative。我们将要在一个数值微分算法中这样做。

函数在一个上微分的结果就是函数图形在该点的斜率。它可以通过计算一点与其附近的一点之间连线的斜率还能容易的估计出来。这里使用了这样的假设：如果两个点足够近，那么函数曲线在这两点之间的弯曲很小，这样我们便有了如下的定义：

```
easydiff f x h = (f(x+h) - f x) / h
```

为了得到精确的近似值，h的取值要非常小。不行的是，如果h太小那么f(x+h)和f(x) 就太接近了，相减时的舍入误差可能会掩盖结果。h到底应该取什么值才合适呢？一个解决这种困境的方法是使用从一个合理的值开始逐渐减小的h值计算一个近似值序列。这个序列应该收敛到导数，但是由于舍入误差的存在，会使结果变得不可救药的不精确。如果把(within eps)用来选择第一个足够接近的近似值，那么舍入误差对结果的影响就会大幅降低。我们需要一个函数来计算这个序列：

```
differentiate h0 f x = map (easydiff f x) (repeat halve h0)
halve x = x/2
```

其中h0是h的初始值，相邻的值通过减半得到。有了这个函数，任何一点的导数都可以如下这样计算：

```
within eps (differentiate h0 f x)
```

即使是这个结果也不能令人满意，因为近似值序列收敛地非常慢。这里一个小数学技巧可以用得上。序列的元素可以这样表示：

```
精确值+ 一个关于h的误差
```

理论上，误差项与h的某次幂成正比，因此其随h的减小而减小。设精确值为A，误差为$B * h^n$。由此，可以使用计算下一个近似值所用h值的2倍的h值计算每个近似值，任何两个相邻的近似值都可以表示为：

$$a(i) = A + B * 2^n * h^n$$

and $a(i+1) = A + B * h^n$

现在误差就可以消去了，我们得到：

$$A = \frac{a(i+1) * 2^n - a(i)}{2^n - 1}$$

当然，因为误差仅仅是h的某此幂的一个近似，所以这个结论也是一个近似，但是是一个更好的近似。这一改进可以通过下面的函数应用到所有的近似值对：

```
elimerror n (cons a (cons b rest)) =
   = cons ((b *(2**n)-a)/(2**n-1)) (elimerror n (cons b rest))
```

从一个近似值序列中消掉误差得到一个收敛快得多的序列。

再使用elimerror之前还有一个问题–要知道n的值。这个值通常很难预测，但是却很容易衡量。不难证明，下面的函数可以正确地估计它，但我们在此不给出证明。

```
order (cons a (cons b (cons c rest))) =
    = round (log2((a-c)/(b-c) - 1))
round x = 离x最近的整数值
log2 x = x 以2 为底的对数
```

现在一个改善近似值序列的通用函数可以定义为：

```
improve s = elimerror (order s) s
```

函数f的通过使用improve像下面这样更有效地计算：

```
within eps (improve (differentiate h0 f x))
```

improve只适用于由一个不断减半的参数h构成的近似值序列。但是它作用于这样一个序列得到的结果还是这样的序列！这意味着近似值序列可以被优化多次。每次优化都有一个不同的误差项被消除，近似值序列收敛也越来越快。因

此，可以非常高效地计算导数：

```
within eps (improve (improve (improve (differentiate h0 f x))))
```

从数值分析的角度说，这像是一举一个四阶方法，可以快速给出精确结果。甚至可以定义：

```
super s = map second (repeat improve s)
second (cons a (cons b rest)) = b
```

这里使用repeat improve 周期来得到一个不断优化的近似值序列构成的序列，然后提取每个近似值序列的第二个元素构造一个新的近似值序列(已经证明第二个值是最好的—它比取首元素更精确而且无需额外的计算)。这个方法的确很复杂—它在不断计算近似值的同时使用了越来越好的数值算法。可以这样相当有效率地计算导数：

```
within eps (super (differentiate h0 f x))
```

这可能有点杀鸡用牛刀了，但重点是要说明即使是复杂如super的算法在使用惰性求值模块化后也可以很容易地表示。

4.3 数值积分

这一节中我们讨论的最后一个例子是数值积分。问题可以被很简单地表述：给定一个参数为一个实数返回一个实数的函数f，和两个端点a和b，估算函数f曲线下方在两个端点之间部分的面积。最简单的办法就是假定f趋向于直线，这时面积便可以表示为：

```
easyintegrate f a b = (f a + f b)(b-a)/2
```

不幸的是除非a和b非常近，否则这样的估计是非常不精确的。一个更好的估计是把从a到b的积分分成两段，分别计算每块的面积然后加起来。使用上面的对第一个近似值的公式，我们可以定义一个越来越好的积分结果近似值构成的序列，每一项都是前面一项各部分二分再加起来。这个序列可以使用如下函数得到：

```
integrate f a b = cons (easyintegrate f a b)
                       (map addpair (zip (integrate f a mid)
                                         (intagrate f mid b)))
其中mid = (a + b)/2
```

zip是另一个标准列表操作函数。它以两个列表作为输入，输出一个序对的列表，每个序对都包含两个列表的相应元素。因此第一个序对包含第一个列表的第一个元素和第二个列表的第一个元素，依此类推。zip可以这样定义：

```
zip (cons a s) (cons b t) = cons (pair a b) (zip s t)
```

在integrate中，zip生成两个子区间上相应积分近似值的序对构成的列表，然后addpair把序对的元素相加得到一个原积分的近似值序列。

实际上这个版本的integrate是很没有效率的，因为它不断重复计算f的值。就像给出的那样，easyintegrate在a和b处计算f的值，然后递归调用integrate时还要重新计算它们。同理，(f mid) 每次递归时计算。因此最好使用下面这个不会重复计算f的版本：

```
integrate f a b = integ f a b (f a) (f b)
integ f a b fa fb = cons ((fa+fb)*(b-a)/2)
                        (map add apair (zip (integ f a m fa fm)
                                            (integ f m b fm fb)))
其中m = (a+b)/2
    fm = f m
```

如同上节的defferetiate一样，integrate生成了一个越来越好的积分近似值构成的无限列表。因此可以写出求任意精度积分的程序：

```
within eps (integrate f a b)
relative eps (integrate f a b)
```

这个积分算法有和上一小节第一个微分算法一样的缺点—收敛太慢。同样，它也可以优化。计算序列的第一个近似值只使用了两个相距为(b-a)的点(通过easyintegrate)，第二个近似值用到了中点，因此相邻两点间距仅为(b-a)/2。

第三个近似值在每一半上都应用这个算法，所以相邻两点的间距仅为(b-a)/4。很明显近似值相邻两点的间距是前一个的一半。这个间距记为“h”，这个序列就可以使用上节的“improve”函数进行优化了。我们可以很快写出快速收敛的积分近似值序列，例如：

super (integrate sin 0 4)
improve (integrate f 0 4)
其中f x = 1/(1+x*x)

(后一个序列是计算pi/4的八阶方法。第二个近似值只需要计算五次f的值就可以得到五位准确数字)。

这一节，我们用函数式编程实现了几个数值算法，并用惰性求值作为黏合剂将部件黏合起来。多亏了惰性求值，我们才能以一种新的方式对它们进行模块化，才能写出像within、relative和improve这么有用的函数。通过把这些部件以不同地方式组合在一起，我们简单明了地编写出了几个相当不错的数值算法。

5 一个来自人工智能的例子

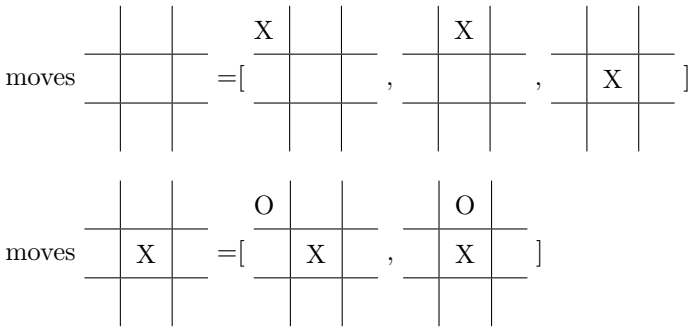
我们已经讨论了函数式编程的强大威力主要是因为它提供了两种新的黏合剂：高阶函数和惰性求值。这一节我们举一个人工智能领域的更大的例子来展示使用这两种黏剂能如何简捷地编写这个程序。

我们选取的例子是alpha-beta“启发式探索”，一个用来评估玩家所处形势的算法。该算法预测游戏的可能发展，同时可以避免探索无意义的方向。

游戏局势用“position”类型的对象表示。这个类型随游戏不同而不同，我们对此不作任何假设。必须有途径知晓在一个position外可以采取的行动：假设有一个函数

moves: position $->$ listof position

它以一个游戏局势为参数，返回一个由此position出发一步可到达的position列表。以圈叉游戏(tick-tack-toe)/footnote在井字形九格中各画0或X, 先将三个连成一行者胜为例



这假定了总是可以通过当前position知道该哪位玩家的回合了。在noughts and crosses 中可以通过数圈和叉的数量确定，在类似象棋这样的游戏中，要在position类型中显式地包含这一信息。

给定了moves，第一步就要建立一棵决策树。该树的节点用position来标记，而其子节点用此节点一步可到达的position标记。就是说，如果一个节点用position p标记，则其子节点用(moves p)中的position来标记。如果游戏可以永远进行下去而没有一方会获胜，那么决策树完全可能是无限的。决策树和我们第二章讨论的树完全类似－每个节点包含一个标签和一个子节点的列表。因此可以使用相同的数据结构来表示它们。

决策树通过重复利用moves来构建。从root position开始，moves被用来根节点的子节点，然后moves又被应用到子节点生成子节点的子节点，依此类推……。这个递归模式可以表示为一个高阶函数：

```
reptree f a = node a (map (reptree f) (f a))
```

用这个函数可以定义另一个函数,可以从特定positon出发构造决策树:

```
gametree p = reptree moves p
```

如图1所示。这里的高阶函数(reptree)和上节中用以产生正在无穷列表的repeat类似。

alpha-beta算法从一个position 出发判断游戏的发展是有利还是不利。然而为了做到这点，必须在不考虑下一个position的情况下粗略估计一个position的价值。在预测受限制的时候，就必须使用这个“静态评价”了，它还可以用来对算法进行先期引导。静态评估是从计算机的角度出发评价一个positon优劣的结

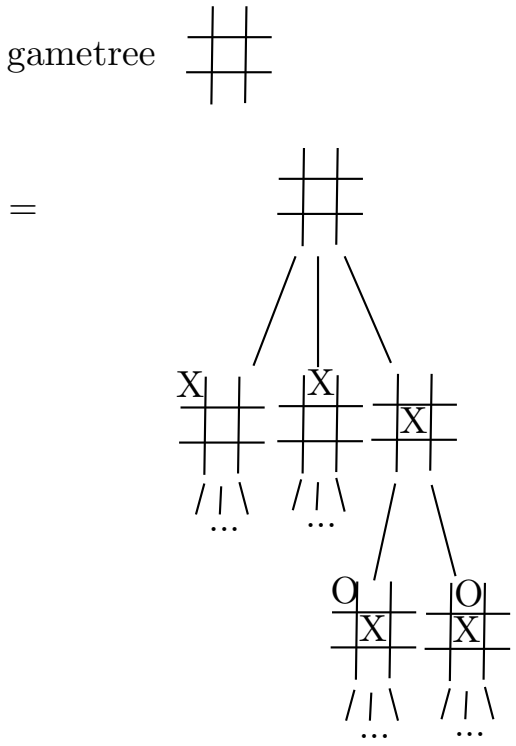


Figure 1: 一个博弈树的例子

果(假设是人机对弈)。结果越大，形势对计算机越有利。结果越小越糟。此函数最简单的版本是计算机获胜时返回1，失败时返回−1，其它情况返回0。在实际中，静态评估函数会衡量各种使局势“看起来不错”的因素，比如物质优势或是象棋中对中心的控制。假设我们有了这样一个函数：

$$\text{static} : \text{position} -> \text{number}$$

既然博弈树是(treeof position)，那么可以通过(maptree static)函数将其转化为(treeof number)，该函数静态评估树上所有的positon（也许是无穷多个）进行评估。此处使用了第二节定义的maptree函数。

有了这样一个静态估值树，各个position的值究竟应该是多大？特别是根position应该赋什么值？不应该是其静态值，因为那只是粗略猜测。一个节点的真值应该由其子节点确定。为了做到这点，要假定每位游戏者均会选择对自己最有利的行动。请记住值越高意味着对计算机越有利，很明显当计算机从任意一个position出发时，均会选择通往具有最大真值子节点的行动。同样，对方会选择通往具有最小真值的子节点的行动。假设计算机和对手轮流行动，轮到计算机时用函数maximise，轮到对手时用minimise函数：

```
maximise (node n sub) = max (map minimise sub)
minimise (node n sub) = max (map maximise sub)
```

这里max和min是作用于元素为数字的列表，分别返回最大和最小的元素。这个定义并不完整，它们会永远递归下去—没有给出边界条件。我们必须定义没有胜利者的结点的值，并且我们将其作为该结点（其标签）的静态评估。因此，在已经有玩家获胜或没有后继结点时该静态评估被使用。maximise和minimise的完整定义为：

```
maximise (node n nil) = n
maximise (node n sub) = max (map minimise sub)
minimise (node n nil) = n
minimise (node n sub) = min (map maximise sub)
```

按照这个步骤甚至可以写出一个函数，它接受一个position，返回其真实值。像这样：

```
evaluate = maximise . maptree static . gametree
```

这个定义有两个问题。首先，它不能处理无限树。Maximise将持续递归直到找到一个没有子树的结点— 树的尽头。如果没有结束，Maximise就不会返回结果。第二个问题是相关的—即使是有限博弈树（）也可以相当大，试图评估整棵树是不现实的（搜索必须被限制在接下来的几步里）。这可以通过修剪树到固定深度解决：

```
prune 0 (node a x) = node a nil
prune n (node a x) = node a (map (prune (n-1)) x)
```

(prune n) 把一棵树离根结点远于n的结点都砍掉。如果一棵博弈树被修剪了，maximise会在深度为n的结点强制使用静态评估而不是进一步递归。因此评估可以定义为：

```
evaluate = maximise . maptree static . prune 5 . gametree
```

这个定义向前看5步。

前面的开发过程中我们已经使用了高阶函数和隋性求值。高阶函数reptree和maptree允许我们轻松构建和维护博弈树。更重要的是，隋性求值确保了我们可以以这种方式模块化evaluate. 因为博弈树具有潜在无穷多的结果，因此没有隋性求值程序永远不会结束。不能只写作

```
prune 5 . gametree
```

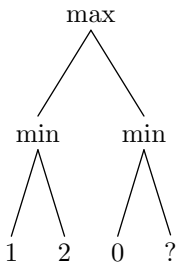
我们将不得不将两个函数整合到一个只构建前5层树的函数。更糟糕的是，即使是前5层也可能太大了以至于不能同时保持在内存里。在我们的程序中，函数

```
maptree static .prune 5 . gametree
```

只构建了maximise需要的部分。因为一旦maximise用完就可以丢掉（被垃圾收

集器回收），所以不可能整棵树都在内存中。某一时刻只有树的一小部分不内存中。因此这个隋性程序是有效率的。这种效率依赖于maximise(组合链上的最后一个函数)和gametree(第一个)的相互作用，没有隋性求值，只能通过把链上的所有函数组合成一个大的来实现。这严重降低了模块化，但这也正是通常的做法。我们可以通过修补每个单独部件来优化评估算法，这相对容易些。一个传统程序员必须把整个程序作为一个整体修改，这要困难的多。

目前为止我们仅描述了简单的maximise和minimise. alpha-beta算法的核心是不用观察整棵树就可以求值maximise和minimise。考虑这棵树： 很奇怪，评



估这棵树并不需要知道问处的值。左面的最小值为1，右面显然是个小于或等于0的值。因此这两个最小值的最大值必然为1。这一观察结果可以一般化并内建到minimise和maximise中。

第一步是将maximise分解成一个max作用于一个数字列表；就是把maximise拆分成:

$$\text{maximise} = \text{max} \cdot \text{maximise'}$$

(minimise以类似的方式拆分。由于minimise和maximise完全对称，我们将只maximise，而假定minimise以类似的方式处理)。这样拆分之后maximise可以使用minimise'而不是minimise本身来发现minimise会对哪些数求最小值。可能不用察看就丢弃某些数。拜隋性求值所赐，如果maximise不必查看列表中所有的数，它们中的一些便不会被计算，这是对计算机时间的潜在节约。

从maximise中"归约出"max是很容易的，有：

$$\text{maximise'}\ (\text{node n nil}) = \text{cons n nil}$$

```
maximise' (node n l)  =  map minimise l
                      =  map (min . minimise') l
                      =  map min (map minimise' l)
                      =  mapmin (map minimise' l)
```

其中mapmin = mapmin

因为minimise'返回一个数的列表，其最小值是minimise的结果，(map minimise' l) 返回一个列表的列表。maximise'应该返回每个列表最小元素构成的列表。但只有这个列表的最大值才有用。我们应该重新定义一个版本的mapmin可以忽略最小值无关紧要的列表。

```
mapmin (cons nums rest) =
        = cons (min nums) (omit (min nums) rest)
```

函数omit传递一个"潜在的最大值" —当前的最大值并忽略比它小的值。

```
omit pot nil = nil
omit pot (cons nums rest) =
        =  omit pot rest,                              if minleq nums pot
        =  cons (min nums) (omit (min nums) rest),     otherwise
```

minleq接收一个列表和一个潜在的最大值，若列表的最小值小于等于这个最大值就返回真。这样，就不必查看整个列表了！如果列表中有任何元素小于或等于这个潜在的最大值，则那么整个列表的最小值一定也是的。此元素后面的元素都无关紧要了—它们像上面例子中的问号一样。因此minleq可以这样定义：

```
minleq nil pot = false
 minleq (cons num rest) pot   = true,              if num<=pot
                              = minleq rest pot,   otherwise
```

这样定义了maximise和minimise之后很容易就可以给出新的评估函数：

```
evalute = max . maximise' . maptree static . prune 8 . gametree
```

多亏了隋性求值，maximise'少扫描树意味着整个程序将运行得更有效率，就

像prune只查看树的一部分使程序能得以结束一样。maximise的优化，尽管很简单，却可以对评估速度产生戏剧性效果，并可以使评估程序可以向前看得更远。

还可以对评估函数做其它优化。比如，上述的alpha-beta算法在只有在归佳行动优先考虑时才会工作地最好，因为一旦发现了一个好算法那么比它差的算法便没必要考虑了，除非能证明对手至少有一种更好的回应。因此可能会希望将每个结点的子树排序，当计算机走的时候最高值放在第一，否则最低值排在第一。可以用下面的函数实现：

highfirst (node n sub) = node n (sort higher (map lowfirst sub))

lowfirst (node n sub) = node n (sort (not . higher) (map highfirst sub))

higher (node n1 sub1) (node n2 sub2) = n1>n2

其中sort 是一个通用的排序函数，评估函数现在可以定义为:

evaluate = max . maximise' . highfirst . maptree static . prune 8 . gametree

为了限制搜索，可以只考虑双方的前三个最佳行动。实现这个程序只要把highfirst替换为(taketree3 . highfirst),其中：

taketree n = redtree (nodett n) cons nil

nodett n label sub = node label (take n sub)

taketree把树中的所有结点替换为最多有n个子结点的结点，使用函数(take n)返回这个列表n个元素的第一个（或更少如果列表元素少于n个）。

另一个改进是重定义prune.上面的程序即使在非常机动的情况下也会向前看固定的深度— 例如在国际象棋里当皇后受到威胁的时候可能不必向前看了。通常定义特定的"机动"形势，在这些形势下不再向前看。假设函数"dynamic"识别这类形势，我们中需要给prune加一个分支等式就可以了：

prune 0 (node pos sub) = node pos (map (prune 0) sub), if dynamic pos

在这样的模块化程序中做这些改变是很被容易的。如前所述，因为程序的效率关键依赖于链中最后一个函数maximise和第一个函数gametree之间的交互作用，没有隋求值只能写成一整个程序。这样的程序难以编写，不好修改前并且晦涩难懂。

6 结论

本文中，我们指出，模块化的成功编程的关键。想要提高生产力的统计编程语言必须良好地支持模块化编程。但是新的作用域规则和分块编译机制是不够的— ”模块化”不仅仅意味着模块。我们能拆分一个问题的能力直接取决于我们将解决方案粘起来的能力。编程语言必须为模块化编程提供好的粘合剂。函数式编程语言提供了两种粘合剂：高阶函数和隋性求值。使用这些粘合剂可以以新的、今人激动的方式模块化程序，这点我们已经举过很多例子了。更小更通用的模块可以被广泛重用，使后续编程更简单。这解释了为什么函数式的程序比传统程序更容易编写。这也为函数式程序员提供一个追求的目标。如果程序的任何一部分杂乱或复杂，程序员就应该试着将其模块化成通用的部件。应该试图以高阶函数和隋性求值为之。

当然，我们不是首先指出高阶函数和隋性求值的力量和优雅的人。比如，Turner展示两者如何在产生化学结构的程序里大展身手[Tur81]. Abelson和Sussman强调流（隋性列表）是构建程序的强有力工具[AS86]。Henderson使用流构建函数式操作系统[P.H82]。本文的主要贡献是断言了模块化本身就是函数式编程语言威力的关键。

这与当前关于隋性求值的论战了有关系。一些个认为函数式语言应该是隋性的，其他人则认为不然。一些人走折中路线，中提供了隋性列表，同学特殊的结构来构造他们（像在Scheme[AS86]中那样）。本文提供了更深入的证据说明隋性求值是如此重要，不能降为二等公民。它可能是函数式程序员所拥有的最强大的粘合剂。人们不能阻碍一个如此重要的函数的使用。